Writing a Simple Kernel Module

25 ottobre 2008
Linux Day

Sommario

In questo breve articolo verrà descritto come scrivere un semplice device driver. In particolare verrà creato uno speciale device *architecture independent* il che prevede la definizione di uno specifico device file. A tale device verranno associate le principali funzioni POSIX e quelle di accesso al filesystem virtuale */proc*.

Per verificare quanto detto, i test verranno fatti su architettura x86 (PC) e AXIS CRIS (FOX Board). Il testo di riferimento è [1].

1 Introdizione

Il modulo che verrà scritto in questo breve articolo è *architecture independent* in quanto esso principalmente non farà altro che creare un buffer il quale potrà essere gestito (riempito, modificato, ecc) dallo user-space accedendo direttamente dal device file o dal filesystem */proc* con diversi permessi. Proprio perchè non verrà toccato nulla a livello hardware, questo modulo è può essere compilato ed utilizzato su qualsiasi architettura. Gli argomenti trattati sono:

- realizzazione di un semplice device driver con relativo file di interfaccia a caratteri
- realizzazione di uno o più file virtuali per l'interfaccia */proc*
- qualche test sulla temporizzazione/schedulazione

Il kernel preso in considerazione per fare il test è il 2.6.13.1 per sistemi x86 e 2.6.15 per sistemi CRIS.

Scelte preliminari

La realizzazione del modulo è, ovviamente, arbitraria. Le scelte fatte per l'implementazione sono le seguenti:

nome — charbuf, identificato dalla macro `DEVICE_NAME` (i suffissi `"charbuf"` o `"cb"` indicheranno funzioni, variabili o altro relative a questo modulo)

major number — 249, definito da `CHARBUF_MAJOR`. Per scegniere il major number, assicurarsi che il numero non sia già usato da un altro dispositivo consultando il file *Documentation/devices.txt*

minor number — progressivo a partire da 0. In questo primo esempio è descritto la manipolazione di un solo device

device file — `/dev/cb0`, ma in realtà il nome e la localizzazione del file non hanno molta importanza

Files

charbuf.c — file principale che raccoglie la quasi totalità del codice vero e proprio del modulo

charbuf.h — file di intestazione in cui sono definite le principali macro, la struttura dati del modulo e le funzioni di I/O sul file di dispositivo implementate

cb_ioctl.h — definizione delle funzioni per la chiamata di sistema `ioctl()`.

2 Funzionamento tipico di un char device

Un device driver che prevede l'esistenza di un file di dispositivo specifico, mette a disposizione dell'utente una serie di funzioni (POSIX) per il menagement del dispositivo stesso. Queste funzioni sono tipicamente le `open`, `read`, `write`, ecc accessibili tramite le librerie specifiche del sistema operativo. Quando un processo vuole accedere ad un file di dispositivo, occorre eseguire una `open`[1] tale per cui il processo ottiene il così detto descrittore del file (file descriptor) il quale non è altro che un numero che identifica univocamente il file aperto. Tutte le altre chiamate di sistema vengono eseguite passando uno specifico descrittore come parametro.

Quando una chiamata di sistema viene invocata (tipicamente dal processo in spazio utente), il sistema operativo sceglie la funzione corretta in funzione del file aperto. É chiaro quindi che ogni device driver, che sia a blocchi o caratteri, implementa queste funzioni in modo a se stante e saranno quindi specifiche per il tipo di dispositivo. Le funzioni sono accessibili mediante la struttura dati `file_operations`; questa struttura è definita in *linux/fs.h*.

3 Implementazione delle funzioni di base

Ogni modulo deve avere due funzioni specifiche per l'inizializzazione in fase di caricamento del modulo e una per l'unload del modulo stesso; queste funzioni sono rispettivamente `charbuf_init_module` e `charbuf_cleanup_module` caricate in fondo al modulo tramite le macro seguenti:

```
module_init(charbuf_init_module);
module_exit(charbuf_cleanup_module);
```

Le operazioni sul file di dispositivo implementate sono:

```
struct file_operations charbuf_fops = {
  .owner = THIS_MODULE,
  .open  = charbuf_open,   // funzione di apertura
  .write = charbuf_write,  // funzione di scrittura
  .read  = charbuf_read,   // funzione di lettura
  .ioctl = charbuf_ioctl,  // funzioni speciali
};
```

Per quanto riguarda le operazioni sui file virtuali in */proc*, queste sono solo funzioni di read e write che verranno definite con specifiche funzioni.

3.1 Generals

All'inizio del modulo potranno essere trovate le seguenti definizioni e macro:

`MODULE_LICENSE("Dual BSD/GPL");` indica la licenza del modulo

`MODULE_VERSION("0.4");` indica la versione del modulo

`static unsigned long bufdimension = DEF_BUF_DIMENSION;` dimensione iniziale di default del modulo

`module_param(bufdimension, ulong, S_IRUGO|S_IWUSR);` parametri del modulo (vedere [1, cap2])

`MODULE_PARM_DESC(bufdimension, "User paramether of initial buffer dimension");` descrizione del modulo

`struct charbuf_dev *charbuf_device;` buffer che contiene la struttura di memoria del driver.

L'utilizzo di una struttura per gestire l'area di memoria allocata dal modulo è solo una scelta arbitraria. La struttura in questione è definita in *charbuf.h*:

[1] In realtà la `open` viene performata all'apertura di un qualsiasi file, non necessariamente di dispositivo, ma anche file lineari, socket, pipe, ecc

```
struct charbuf_dev {
   char *charbuffer;    ///è il buffer vero e proprio contenuto nel kernel
   unsigned int size;   ///dimensione netta del device
   unsigned int index;  ///attuale posizione all'interno del buffer,
                        ///necessario per l'append dei dati
   struct semaphore sem;///mutual exclusion semaphore - serve! vedi LDD3 cap3
   char    struct cdev cdev; ///necessario per la gestione dei dispositivi
};
```

3.2 /dev/cb0

Seguirà ora una breve descrizione delle funzioni del modulo. Questo modulo non farà altro che creare un buffer in spazio kernel accessibile in read/write dallo user-space.

3.2.1 charbuf_init_module

Questa è la funzione di inizializzazione del modulo il cui prototipo deve essere:

```
int charbuf_init_module(void)
```

In questa prima fase, che non prevede ancora l'inizializzazione dell'interfaccia */proc*, lo scopo sarà unicamente preparare la memoria necessaria all'utilizzo del modulo.

La prima cosa da fare è registrare il dispositivo a caratteri mediante la funzione

```
register_chrdev_region(dev,1,DEVICE_NAME)
```

dove:

`dev` è un valore calcolato tramite la macro `MKDEV` il cui risultato dipende dal major e minor number;

`1` è un counter e indica quanti device vogliamo creare. Per esempio se i device da creare fossero *cb0*, *cb1*, *cb2*, questo valore sarebbe 3.

`DEVICE_NAME` è la stringa che identifica il nome del modulo, `charbuf` in questo caso. Questa stringa apparirà, una volta caricato il modulo, nel file */proc/devices* con il relativo major number.

Se il device viene registrato correttamente si procede con l'allocazione della memoria con il comando

```
charbuf_device=kmalloc(sizeof(struct charbuf_dev), GFP_KERNEL);
```

dove il buffer in questione è una variabile globale e la funzione `kmalloc` non fa altro che chiedere al sistema un'area di memoria a cui far puntare la struttura. Questa funzione è simile alla funzione `malloc` del C, con la differenza che richiede un parametro in più, in questo caso `GFP_KERNEL`. Questo parametro indica al kernel che può cercare aree di memoria anche non contigue (ossia frammentate); ciò implica chiaramente l'uso della MMU. Per maggiori informazioni vedere [1, cap8].

Nota: in questo caso viene creato un solo device (*/dev/cb0*), ma se volessimo crearne di più dovremmo allocare differenti aree di memoria per ogni file di dispositivo. Normalmente ciò viene fatto moltiplicando per il numero di dispositivi il valore restituito dalla `sizeof(struct charbuf_dev)`.

Ora il modulo è caricato, ma occorre creare il buffer nella struttura `charbuf_dev` il quale non deve eccedere dal valore `MAX_BUF_DIMENSION` definito nel file *charbuf.h*. Seguono quindi le istruzioni:

```
init_MUTEX(&charbuf_device->sem);
charbuf_setup_cdev(charbuf_device,0);
```

dove la prima inizializza i semafori che serviranno alle funzioni di sistema per evitare le race conditions ([1, Cap5-Concurrency and Race Conditions]), mentre la seconda inizializza il dispositivo a caratteri linkando anche le varie funzioni necessarie per il menagement (read, write, ecc).

Questa funzione viene passata alla macro `module_init()` in modo che il sistema la chiami a fronte del caricamento del modulo per esempio con i comandi `insmod` o `modprobe`.

3.2.2 charbuf_cleanup_module

Questa funzione permette di eseguire alcune operazioni sul modulo in fase di unload (per esempio con il comando `rmmod`). Il prototipo deve essere:

```
void charbuf_cleanup_module(void);
```

e l'implementazione è:

```
void charbuf_cleanup_module(void) {
   dev_t dev_number=MKDEV(CHARBUF_MAJOR, 0);

   charbuf_device->cdev.ops=NULL;
   cdev_del(&charbuf_device->cdev);

   if(charbuf_device->charbuffer)
     kfree(charbuf_device->charbuffer);
   if(charbuf_device)
     kfree(charbuf_device);
   charbuf_device=NULL;

   unregister_chrdev_region(dev_number, 1);
}
```

Si noti che l'istruzione `cdev_del(&charbuf_device->cdev);` è fondamentale perchè si supponga di togliere il modulo dal kernel, ma di non cancellare il file di dispositivo */dev/cb0*; ora se si cerca di aprire il file verrebbe generato un SEGFAULT dal programma che cerca di accedere ad esso. Con questo accorgimento invece se provassimo ad accedere al modulo si avrebbe:

```
echo X > /dev/cb0
bash: /dev/cb0: No such device or address
```

mentre l'istruzione `charbuf_device->cdev.ops=NULL;` è accessoria, ma è buona norma prevederla.

Le istruzioni successive liberano la memoria del device con una sequenza dipendente dell'incapsulamento dei puntatori.

Questa funzione viene passata alla macro `module_exit()` in modo che il sistema la chiami a fronte della disinstallazione del modulo per esempio con il comando `rmmod`.

3.2.3 charbuf_open

La funzione *open* è la funzione più importante in quanto viene sempre eseguita almeno una volta dal processo padre e configura l'area di memoria accessibile per la altre funzioni. Il prototipo deve essere il seguente:

```
int charbuf_open(struct inode *inode, struct file  *flip);
```

tramite `inode` viene individuato il file aperto dallo user-space, ma la cosa principale è la struttura `flip` che permetterà di accedere alla corretta area di memoria del dispositivo:

```
int charbuf_open(struct inode *inode, struct file  *flip) {
   struct charbuf_dev *dev;
   printk(KERN_ALERT "Open charbuffer\n");
   dev = container_of(inode->i_cdev, struct charbuf_dev, cdev);
   if(!dev)
      printk(KERN_ALERT "WARNING: charbuffer device is null!\n");
   flip->private_data = dev;

   if((flip->f_flags & O_APPEND) == O_APPEND) { //APPEND
      printk(KERN_ALERT "Append.\n"); //segnalo la cosa, ma non faccio nulla
   }
```

Se il file è aperto per la sola scrittura, va resettato e `.index` viene posto a 0:

```
    else if((flip->f_flags & O_ACCMODE) == O_WRONLY) {
       if(down_interruptible(&dev->sem))
          return -ERESTARTSYS;
       printk(KERN_ALERT " Open: reset buffer.\n");
       memset(dev->charbuffer,0, dev->size);
       dev->index=0;
    }
    up(&dev->sem);
    return 0;
}
```

come detto si vede che l'istruzione `container_of` pesca dalla struttura `inod` l'indirizzo della struttura dati del modulo; in altre parole l'indirizzo di `charbuf_device` in questo caso (in quanto abbiamo un solo file di dispositivo). Questo viene poi copiato nella struttura relativa al file `flip->private_data = dev;` in modo che le altre chiamate di sistema possano puntare alla stessa struttura dati.

Le istruzioni successive non fanno altro che verificare la modalità di apertura del file. Se `O_WRONLY` il file è aperto in sola scrittura e verrà completamente azzerato. Si noti come l'azzeramento del file comporti l'utilizzo della funzione `down_interruptible()` la quale impedisce ad un altro processo, che magari sta per accedere al dispositivo, di operare funzioni di scrittura o lettura sulla stessa area di memoria. A questo punto è necessaria anche la funzione `up(&dev->sem);` per sbloccare l'accesso al file.

Se tutto va a buon fine la funzione restituisce 0.

3.2.4 charbuf_write

Questa funzione permette di scrivere nel buffer. La modalità di scrittura prevede le seguenti operazioni:

- scrivere una quantità di dati massima pari alla dimensione corrente del buffer la quale non può essere ridimensionata se si eccede.
- scrivere partendo da una locazione qualsiasi del buffer, ovviamente compresa nella massima dimensione.
- accedere in append mode per aggiungere dati.
- se lo user-space vuole scrivere più dati della dimensione del buffer, il sistema deve scrivere fino alla massima dimensione e restituire un errore.

La funzione *write* restituisce il numero di byte scritti nel buffer del modulo. Il valore 0 è da intendersi come l'impossibilità di proseguire oltre e quindi, in questo caso, il buffer è stato scritto fino alla fine (almeno dal punto di vista del processo user-space). Se il valore è negativo è accaduto un errore. La funzione è la seguente:

```
ssize_t charbuf_write(struct file *flip, const char __user *buf,
                      size_t count, loff_t *f_pos) {
```

inizialmente si inizializzano le variabili di sistema definisco subito qual è il buffer su cui lavorare tramite `*dev = flip->private_data`; questo è possibile farlo grazie a quanto fatto durante la funzione *open*.

```
    struct charbuf_dev *dev = flip->private_data; //carico il buffer charbuf_device
    ssize_t returnvalue=0;
    int err;
```

blocco l'accesso ad altri processi. Se un processo ha già aperto il file, il processo concorrente non può accedervi in quanto il kernel ritorna un errore `ERESTARTSYS`.

```
    printk(KERN_ALERT "Write\n count=%i\n",count);
    if(down_interruptible(&dev->sem)) //blocco l'accesso
      return -ERESTARTSYS;            //da parte di altri processi
```

Viene quindi controllato se la posizione a cui si cerca di scrivere eccede la dimensione del file. Se sì si esce con un errore

```
if(*f_pos >= dev->size)
  {
     returnvalue=-ENOSPC;
     goto exit_write_fcn; //esci dalla funzione write
  }
```

Controllo se il file è aperto in append o no. "Append" equivale anche a un normale write in un punto qualsiasi del buffer

```
if((flip->f_flags & O_APPEND) == O_APPEND) //APPEND
  {
     //se sono gia alla fine del buffer non ha senso proseguire...
     if(dev->index >= dev->size-1)
       {
          returnvalue=-ENOSPC;
          goto exit_write_fcn; //esci dalla funzione write
       }
     else
```

Ridimensionare la variabile `count` per evitare di eccedere la lunghezza del buffer:

```
       if(*f_pos + count + dev->index >= dev->size)
         count = dev->size - (*f_pos + dev->index);
     err = copy_from_user(&dev->charbuffer[*f_pos + dev->index],
                          buf, count);
  }
else //se non è un append allora è un write only (quindi un rewrite)
  {  //il buffer è gia resettato grazie alla funzione open
     if(*f_pos + count > dev->size)
       count = dev->size - *f_pos;
     err = copy_from_user(&dev->charbuffer[*f_pos], buf, count);
  }

if(err)
  {
     returnvalue = -EFAULT;
     goto exit_write_fcn;
  }
```

Alla fine si esce aggiornando tutte le variabili, ossia indico al kernel che la posizione di scrittura all'interno del file è aumentata di `count`, similmente indico al processo che siamo avanzati di `count` e aggiorno anche l'indice. Quindi si esce permettendo nuovamente l'accesso da parte di altri processi.

```
     *f_pos += count; //aggiorno il file pointer
     returnvalue = count; //restituisco il numero di byte copiati
     dev->index += count; //Aggiorno l'index

exit_write_fcn:
     up(&dev->sem);
     return returnvalue;
}
```

3.2.5 charbuf_read

La funzione permette di leggere il buffer da una qualsiasi posizione ammissibile. Quando una *read* viene eseguita, il programma in user-space si aspetta che tale funzione ritorni un valore maggiore o uguale a 0 che indica quanti byte sono stati trasferiti e copiati nel buffer passato dallo user-space (`char __user *buf`). Quando la funzione ritorna 0, il processo interpreta il dato come un "end of file" (*eof*), mentre un valore negativo è da gestire come un errore.

Questa funzione è concepita in modo che:

- venga scritto un buffer in qualsiasi posizione minore o uguale al valore di `index`

- l'append dei dati venga fatto al valore di `index`
- il buffer non sia ridimensionabile oltre la grandezza `size`

```
ssize_t charbuf_read(struct file *flip, char __user *buf,
                     size_t count, loff_t *f_pos) {
```

`dev` deve puntare ai dati del device driver, dati caricati tramite la chiamata *open*.

```
struct charbuf_dev *dev = flip->private_data;
ssize_t returnval=0;

printk(KERN_ALERT "Read.\n");     // per il debug
if(down_interruptible(&dev->sem)) // evita la race concurrency
  return -ERESTARTSYS;
printk(KERN_ALERT "size = %i\nidx  = %i\nf_pos= %i\ncount= %i\n",
       (int)dev->size, dev->index, (int)*f_pos, (int)count);
```

La riga seguente evita di accedere al buffer puntando in un'area non ammessa. `count` vine messo a 0 perchè quando la funzione ritorna 0 indica che il buffer dello user space non può più essere maggiore della dimensione effettiva del buffer

```
if(*f_pos >= dev->size)
  count = 0; //in teoria potrei fare semplicemente 'return 0',
             // ma preferisco fare cosi per avere un feedback di debug
else
```

Si ricorda che `index` è l'indice all'interno del buffer che indica l'attuale posizione a cui sono stati scritti i dati, quindi non bisogna eccedere questo valore. Si noti che mettendo `dev->size` al posto di `dev->index` si genererebbero dei loop infiniti leggendo il device con il comando *cat*.

```
if(*f_pos + count >= dev->index)
  count = dev->index - *f_pos;

printk(KERN_ALERT "copy %i byte...\n", count);
if(copy_to_user(buf,dev->charbuffer + *f_pos, count))
  {
     printk(KERN_ALERT "Error while copy to user space!\n");
     returnval = -EFAULT;
     goto exit_read_fnc;
  }

exit_read_fnc:
*f_pos += count;
returnval=count;
up(&dev->sem);

   return returnval; //se 0 è perchè nulla è copiato e quindi blocco la lettura
}
```

3.2.6 charbuf_ioctl

A questo punto il driver sarebbe già testabile. Nonostante ciò viene introdotta subito la funzione di *ioctl*. Questa funzione permette di eseguire operazioni speciali sul file. Tali operazioni devono essere notee a disposizione allo sviluppatore user-space e ognuna di esse è definita da un numero. La funzione user-space di *ioctl* (accessibile mediante *sys/ioctl.h*) ha il seguente prototipo:

```
int ioctl (int __fd, unsigned long int __request, ...)
```

ossia richiede ovviamente un file descriptor e un valore che indica la richiesta da eseguire. Il parametro che segue può essere passato in vari modi. In questo modulo, e in questa funzione specialmente, verrà passato sempre un puntatore. Per capire dove vanno a finire questi parametri nella corrispondente funzione del kernel si osservi il prototipo della funzione *ioctl* del modulo:

```
int charbuf_ioctl(struct inode *inode, struct file *filp,
                  unsigned int cmd, unsigned long arg) {
```

dove `cmd` è il comando (o richiesta) `__request`, mentre `arg` è il terzo parametro della funzione *ioctl* dello user-space. I comandi di ogni device devono però essere noti a priori e vengono normalmente posti in un opportuno file header disponibile anche per la programmazione user-space. Nel caso in esame si è scelto di definirle nel file *cb_ioctl.h*. Normalmente i comandi si dividono in *Query*, *Set*, *Get* e *Tell* (per maggiori informazioni si rimanda sempre a [1, pag.135]). charbuf implementa nello specifico quattro funzioni di esempio definite come segue:

```
#define CB_IOCTL_QSIZE   _IO(CB_IOC_MAGIC, 0)
#define CB_IOCTL_QINDEX  _IO(CB_IOC_MAGIC, 1)
#define CB_IOCTL_SSIZE   _IOW(CB_IOC_MAGIC, 2, int)
#define CB_IOCTL_SBUF    _IOW(CB_IOC_MAGIC, 3, struct cb_buf*)
```

Queste macro si basano su un "numero magico" `CB_IOC_MAGIC=0x05`. La scelta di questi valori non è del tutto arbitraria, ma dipenderebbe dai driver/moduli già implementati. Ad ogni modo, la prima funzione restituirà la dimensione massima del buffer e la seconda restituirà l'indice (ossia la quantità di dati attualmente scritta). La terza e quarta invece eseguono una scrittura: in particolare la `_SSIZE` setterà il valore del buffer, mentre `_SBUF` il suo contenuto. Per rendere le cose più interessanti si vuole anche fare in modo che solo root possa settare più del valore di `dev->index`, mentre l'utente non privilegiato dovrà limitarsi a quantità minori o uguali di `dev->index`.

Come al solito si comincia con la definizione dei file e l'identificazione del buffer del modulo:

```
struct charbuf_dev *dev = filp->private_data;
int err=0, retval=0;
int i,iaux; //integer ausiliary value
char *auxbuf;

printk(KERN_ALERT "Richiesto accesso a ioctl func (%i)...\n", cmd);
```

Controllo che il comando richiesto dallo user-space sia per questo tipo di dispositivo e soprattutto che esista:

```
if(_IOC_TYPE(cmd) != CB_IOC_MAGIC) //controllo che sia il comando richiesto
  return -ENOTTY;
if(_IOC_NR(cmd) > CB_IOCTL_MAXNR) //se eccedo con i comandi non va bene.
  return -ENOTTY;
```

Tramite `access_ok` posso verificare che il buffer passato sia in lettura e/o scrittura.

```
if(_IOC_DIR(cmd) & _IOC_READ)
  err = !access_ok(VERIFY_WRITE, (void __user *)arg, _IOC_SIZE(cmd));
if (err)
  return -EFAULT;
```

I comandi vengono ora processati

```
 switch(cmd)
   {
    case CB_IOCTL_QSIZE: // restituisco la dimensione
      return dev->size;
      break; //inutile a causa del return

    case CB_IOCTL_QINDEX: // restituisco l'indice del buffer
      return dev->index;
      break;
```

Il nuovo valore di dimensione deve essere settato, ma senza perdere i dati scritti. Per fare questo ci si appoggia a un puntatore ausiliario.

```
        case CB_IOCTL_SSIZE: //setto il nuovo valore
          if(down_interruptible(&dev->sem)) // blocco gli accessi multipli
            return -EBUSY;

          retval = __get_user(iaux, (int __user *)arg); //un solo dato dal buffer
          if(iaux > MAX_BUF_DIMENSION) //il nuovo valore non deve essere enorme
            iaux = MAX_BUF_DIMENSION;
          if(iaux != dev->size) { //se sono uguali non fare nulla
             if(!(auxbuf = kmalloc(iaux, GFP_KERNEL))) {
                  printk(KERN_ALERT "Not enought memory.\n");
                  retval = -ENOMEM;
               } else
               {
                  if(iaux < dev->index)
                    dev->index = iaux;
                  printk(KERN_ALERT "Redimensioning buffer to %i byte throught IO-Ctl\n", iaux);
                  for(i=0; i<dev->index; i++)
                    auxbuf[i]=dev->charbuffer[i];
                  kfree(dev->charbuffer);
                  dev->charbuffer=auxbuf;
               }
             dev->size = iaux;
          }

          up(&dev->sem); // permetto ad altri processi di operare
          break;

        case CB_IOCTL_SBUF: //setta il contenuto del buffer
          if(down_interruptible(&dev->sem))
            return -EBUSY;
```

Per riempire il buffer si utilizza la struttura `struct cb_buf`. Si veda `cb_ioctl.h`.

```
          struct cb_buf *usbuf; //userspace buffer
          retval = __get_user(usbuf, (struct cb_buf __user **)arg);
          if(capable(CAP_SYS_ADMIN)) //se sei amministratore puoi scrivere oltre .index
            {
               if(usbuf->dim > dev->size) //non puoi scrivere oltre .size, però!
                 {
                    usbuf->dim = dev->size;
                    dev->index = dev->size;
                 }
               else
                 dev->index = usbuf->dim;
            }
          else if(usbuf->dim > dev->index)
            {
               usbuf->dim = dev->index;
               printk(KERN_ALERT "Only root can write more then .index\n");
            }
          else
            dev->index = usbuf->dim;

          for(i=0; i<usbuf->dim; i++)
            dev->charbuffer[i] = usbuf->buf[i];

          up(&dev->sem);
          break;
      }

    return retval;
}
```

3.3 Test delle funzioni base

A questo punto il modulo può essere compilato e testato. Per la compilazione si rimanda all'appendice B. Per testarlo occorre tenere sotto controllo il file */var/log/messages* (che intercetta i messaggi del kernel marcati

con `KERN_INFO`) e */var/log/syslog* (che intercetta i messaggi del kernel marcati con `KERN_ALERT`) con il comando

```
# tail -f /var/log/messages
# tail -f /var/log/syslog
```

quindi creare il device file, non necessariamente in */dev* anche se ora verrà creato lì, con il comando:

```
# mknod /dev/cb0 c 249 0
```

si ricorda che "#" posto come primo carattere del comando indica che i comandi devono essere lanciati da root mentre con "$" da utente.

Una volta compilato il modulo lo si carica con il comando *insmod* se non è installato nelle directory di default del kernel (scelta consigliata almeno in fase di debug), altrimenti si usa il tipico comando *modprobe*:

```
# insmod charbuf.ko
```

si ricorda che l'estensione *.ko* è dovuta al kernel 2.6. Se tutto va a buon fine, sulla shell che esegue il comando *tail* di *messages* comparirà:

```
[...] kernel: Initailization of charbuf (major-minor 249-0)
```

Si noti che i tre punti iniziali indicano dei dati che possono variare da distro a distro e anche con le configurazioni del logger; `kernel` indica solo che il messaggio è stato generato dal kernel. Normalmente si leggerà la data e il nome del computer: in seguito non verranno più visualizzati. Nel file */var/log/syslog* si ha invece la nota sull'inizializzazione del buffer:

```
Registration done.
Alloc memory (512 byte)
```

512 è la dimensione di default. Passando il parametro *bufdimension=1000* il parametro deve cambiare appunto in 1000. Se il modulo venisse scaricato, si avrebbe:

```
Unregister char region (charbuf)
```

A questo punto si procede con i test, ovviamente con il device caricato.

open

La funzione *open* si testa semplicemente con *touch*:

```
# touch /dev/cb0
```

Il messaggio di *syslog* è:

```
Open charbuffer
 Open: reset buffer.
```

il che è corretto perchè il device viene sì aperto, ma il buffer appena allocato deve essere resettato, ossia vengono scritti tutti 0 per tutta la dimensione del buffer. `.index` viene quindi posto pari a 0. Ciò viene fatto perchè *touch* apre in "sola scrittura" il device e quindi, secondo il codice scritto, il buffer verrà sempre resettato.

Per testare l'apertura in "sola lettura" si esegue:

```
# < /dev/cb0
```

il quale genererà la sola linea:

```
Open charbuffer
```

write e append

Si supponga di caricare il driver avendo definito la dimensione del buffer pari a 15 byte. Eseguendo:

```
# echo 1234567890 > /dev/cb0
```

syslog restituirà:

```
Open charbuffer
 Open: reset buffer.
Write
 count=11
```

Ovviamente vi è l'apertura del device a cui segue la scrittura di 11 caratteri. 11 e non 10 perchè *echo* invia anche il carattere "return" (`\n`). 11 e non 12 perchè in UNIX il carattere di return è solo `\n` e non `\n\r`.

L'append si verifica con:

```
#echo 123 >> /dev/cb0
```

e *syslog* restituisce:

```
Append.
Write
 count=4
```

Vengono aggiunti in fondo altri 4 byte. Si supponga ora di scrivere più di 15 byte:

```
# echo 12345678901234567890 > /dev/cb0
```

syslog restituisce:

```
Open charbuffer
 Open: reset buffer.
Write
 count=21
Write
 count=6
```

Una volta aperto il buffer in kernel space, le istruzioni da user-space dicono che si vogliono scrivere 21 byte. A questo punto il modulo si accorge che 21 byte sono troppi e setta `count = dev->size - *f_pos = 15`. Quindi alla fine viene ritornato allo user-space il valore 15. *echo* però vedendo che il kernel ha scritto solo 15 byte, reinvia al kernel la richiesta di scrittura di altri 21-15=6 byte da scrivere nel buffer alla posizione `*fpos` che ora vale 15. Quindi la *write* viene richiamata, ma ci si accorge subito che viene eseguito con succerro la seguente parte di codice:

```
if(*f_pos >= dev->size) {
   returnvalue=-ENOSPC;
   goto exit_write_fcn;
}
```

questo implica che la funzione ritorna subito con un valore minore di 0 (in questo caso "errore non specificato"). Per esserne certi è sufficiente lanciare il comando:

```
# echo $?
1
```

Il fatto che echo restituisca 1 e non 0 (cioè un errore) è solo una scelta progettuale arbitraria. Si è verificato ciò che si voleva ottenere. Le stesse prove possono essere fatte in "append":

```
#echo 1234567890 >  /dev/cb0
#echo 1234567890 >> /dev/cb0
```

read

Lettura in blocco del dispositivo:

```
# cat /dev/cb0
123456789012
```

che sertituirà:

```
Open charbuffer
Read.
size = 15
idx  = 13
f_pos= 0
count= 131072
copy 13 byte...
Read.
size = 15
idx  = 13
f_pos= 13
count= 131072
copy 0 byte...
```

Il device viene aperto e letto. *cat* vuole 131072 byte, mentre il buffer è caricato con 13 byte. Il sistema, partendo dall'inizione del buffer (`*fpos=0`), copia 13 byte e lo comunica allo user-space il quale cerca di leggere ancora in quanto si ricorda che lo user-space vede la end-of-file a fronte del ritorno di uno 0. Essendo però giunto alla fine del buffer "utile" il modulo restituisce 0 e il sistema in user-space si ferma.

Per vedere meglio come si comporta una *read* si possono usare i seguenti comandi:

```
# dd if=/dev/cb0 count=2 bs=2
# dd if=/dev/cb0 count=2 bs=2 skip=3
```

Il primo permette di vedere come viene gestito `count` e si nota che letti 4 byte il sistema si ferma. Il secondo invece fa notare come la posizione del file sia diversa da 0 fin dall'inizio (in particolare pari a $2 \cdot 3 = 6$). Ponendo l'opzione *skip* troppo alta si può verificare cosa accade se `*fpos` è troppo alta.

ioctl

Per testare le *ioctl* occorre avere un programma ad hoc. É necessario includere `cb_ioctl.h`, ossia l'header del modulo che definisce i valori e le macro che identificano le funzioni.

```
#include <stdio.h>
#include "cb_ioctl.h"
#include <sys/ioctl.h>
#include <unistd.h> //per le open, close, create, ecc
#include <sys/types.h> //flag x open
#include <sys/stat.h>
#include <fcntl.h>
#define STR "Super classifica show\n"
int valore=10;
struct cb_buf cb = {
   .dim=sizeof(STR),
   .buf=STR
};
char *s;
struct cb_buf *pcb=&cb;
int main(int c, char **v) {
   int descriptor;

   descriptor=open(v[1],O_RDONLY);
   if(descriptor < 0) {
        printf("Error while opening %s\n", v[1]);
        return descriptor;
     }
```

```
    printf("Aperto file dev: %i\n",descriptor);
    printf("Return IO-CTL: %i\n", ioctl(descriptor, CB_IOCTL_QSIZE));
    ioctl(descriptor, CB_IOCTL_SSIZE, &valore);
    ioctl(descriptor, CB_IOCTL_SBUF, &pcb);
    close(descriptor);
    return 0;
}
```

Questo programma apre il dispositivo, legge la dimensione massima, ossia `.size`, setta il nuovo valore massimo e infine scrive una stringa nel buffer. Lanciando il programma due volte da root si ottiene come risposta:

```
# ./access_ioctl /dev/cb0
Aperto file dev: 3
Return IO-CTL: 512
# ./access_ioctl /dev/cb0
Aperto file dev: 3
Return IO-CTL: 10
```

Il sistema sembra funzionare. Nel */var/log/syslog* si ha:

```
Open charbuffer
Richiesto accesso a ioctl func (1280)...
Richiesto accesso a ioctl func (1074005250)...
Redimensioning buffer to 10 byte throught IO-Ctl
Richiesto accesso a ioctl func (1074005251)...
Open charbuffer
Richiesto accesso a ioctl func (1280)...
Richiesto accesso a ioctl func (1074005250)...
Richiesto accesso a ioctl func (1074005251)...
```

Correttamente vengono fatte 3 richieste di cui la prima effettua il ridimensionamento del buffer, mentre la seconda volta che *access_ioctl* non esegue nessun ridimensionamento in accordo con il codice del modulo. Il programma però scrive anche una stringa. Con il solito *cat* si verifica la scrittura del driver che restituisce:

```
Super clas
```

Questa stringa, senza return, è esattamente 10 byte in quanto anche root non può scrivere oltre `.size`. Ora si supponga di scrivere nel buffer 5 byte e di rilanciare il programma, questa volta da utente:

```
# rmmod charbuf
# insmod charbuf.ko
# echo -n 12345 > /dev/cb0
$ ./access_ioctl /dev/cb0
Aperto file dev: 3
Return IO-CTL: 512
$ ./access_ioctl /dev/cb0
Aperto file dev: 3
Return IO-CTL: 10
```

Sembra che sia tutto uguale a prima ma non lo è perchè *cat* questa volta restituirà:

```
Super
```

ossia 5 byte. Mentre in */var/log/syslog* si avrà, oltre alle linee di cui sopra:

```
Only root can write more then .index
```

4 Interfacciamento con /proc

Le funzioni in */proc* vengono sviluppate includendo *linux/proc_fs.h*. Il filesystem virtuale permette al modulo di interfacciarsi con lo spazio utente tramite delle directory e file (virtuali appunto) i quali hanno implementano particolari funzioni sia in scrittura che in lettura. Normalmente un file virtuale non ha bisogno di una struttura coma `file_operation` in quanto ciò che interessa è semplicemente implementare le due funzioni di lettura e scrittura.

4.1 /proc/charbuf

L'implementazione della struttura delle directory di /proc è fatta tipicamente in fase di inizializzazione del modulo. Ovviamente l'albero delle directory (e dei file) deve essere creato in sequenza e liberato in fase di unload del modulo partendo dal file (o drectory) più profondo. Questa prassi è seguita anche in questo nostro modulo e realizzerà una directory */proc/charbuf* con all'interno 3 files: *buffer*, *dimension*, *timerstr*.

dimension "contiene" la dimensione massima del buffer (`.size`) e il valore attuale dell'index (`.index`). Tramite `charbuf_write_procmem_dim` è possibile moltiplicare o dividere la dimensione del buffer.

buffer permette la lettura del buffer. A questo file è asservita la funzione di lettura `charbuf_read_procmem_buf` e la funzione di scrittura `charbuf_write_procmem_buf` che permette di azzerare il buffer.

timerstr funzione di test del timer, non affrontato in questo capitolo.

4.1.1 charbuf_init_module

Al codice definito al capitolo viene aggiunto:

```
proc_charbufdir=proc_mkdir(PROC_CHARBUFDIR,NULL); //creo la subdir in proc
if(!proc_charbufdir) {
   printk(KERN_ALERT "WARNING: proc_charbuf_dir NULL!\n");
} else
{
   proc_charbufdim = create_proc_read_entry(PROC_CHARBUFDIM, /*virtual filename*/
                                            0, /*default mode*/
                                            proc_charbufdir, /*parent dir*/
                                            charbuf_read_procmem_dim, /*funzione scritta da me*/
                                            NULL );
   proc_charbuffer = create_proc_read_entry(PROC_CHARBUFFER,
                                            0, proc_charbufdir,
                                            charbuf_read_procmem_buf, NULL);
   proc_chartmrstr = create_proc_read_entry(PROC_TIMERSTR,
                                            0, proc_charbufdir,
                                            charbuf_read_procmem_tmrstr, NULL);
   if(!proc_charbufdim || !proc_charbuffer || !proc_chartmrstr) {
      printk(KERN_ALERT "WARNING: wrong init /proc filesystem!\n");
   } else
   {
      proc_charbufdim->write_proc=charbuf_write_procmem_dim;
      proc_charbuffer->write_proc=charbuf_write_procmem_buf;
      proc_chartmrstr->write_proc=charbuf_write_procmem_tmrstr;
   }
}
```

Inizialmente viene creata la directpry virtuale *charbuf* e se tutto va a buon fine vengono creati i file virtuali definendo che appartengono alla directory *charbuf* (`proc_charbufdir`). Se vengono creati tutti e tre i file, allora viene anche assegnata ad ognuno di essi la funzione di scrittura. La definizione (e implementazione) di entrambe queste due funzioni non è obbligatoria se non dovessero servire. Per maggiorni informazioni vedere la struttura `proc_dir_entry` in *linux/proc_fs.h*.

4.1.2 charbuf_cleanup_module

In fase di clean-up occorre deallocare le varie entry partendo dagli elementi più nidificati. In questo caso si parte dai file e, una volta rimossi dalla struttura */proc/charbuf*, si rimuove anche la directory *charbuf* come segue

```
remove_proc_entry(PROC_TIMERSTR,    //file name
                  proc_charbufdir); //parent dir
remove_proc_entry(PROC_CHARBUFDIM,  //file name
                  proc_charbufdir); //parent dir
remove_proc_entry(PROC_CHARBUFFER,  //file name
                  proc_charbufdir); //parent dir
remove_proc_entry(PROC_CHARBUFDIR,  //file name
                  NULL); //parent dir, null=/proc
```

4.1.3 Funzioni per `dimension`

Le funzioni implementate sono sia di lettura che di scrittura: `charbuf_read_procmem_dim` e `charbuf_write_procmem_dim`.

charbuf_read_procmem_dim

Questa funzione permette di leggere (o al limite "creare") dei dati da trasferire in fase di accesso al file contenuti in */proc/charbuf/dimension* (definito da `PROC_CHARBUFDIM`). Un esempio di funzione di questo tipo è a [1, pag 85 'Implement file in /proc']. Lo stesso [1] dice che l'esempio è un po' brutto e credo si riferisca la fatto che il sistema non presenta un controllo relativo al buffer che sta scrivendo; buffer che comunque è in user-space quindi non dovrebbe causare gravi problemi di sicurezza.

In questa funzione si andrà quindi a creare prima la stringa (ossia i dati) da trasferire, e solo successivamente verranno copiati nel buffer a poco a poco (byte x byte) in modo da tenere sotto controllo il limite (`count`) del buffer. Altra cosa molto importante è la gestione dell'offset che viene fatta nella fase iniziale. Ciò rende la procedura più completa nell'eventualità che venga letto il file a partire da un punto maggiore del primo byte (`offset>0`).

Ad ogni modo è richiesto che questo file contenga sulla prima linea il valore di `.size` e sulla seconda il valore di `.index`.

```
int charbuf_read_procmem_dim(char *buf, char **start, off_t offset,
                             int count, int *eof, void *data) {
   int numlen=0, len=0, i;
   char num[12]; //dovrebbero bastare 11 byte: "4294967295"+'\n'
   struct charbuf_dev *device=charbuf_device;
```

Anche in questo caso evito le race conditions:

```
   if(down_interruptible(&device->sem))
     return -ERESTARTSYS;
   if(!buf) //se il buffer è nullo è meglio uscire ;) ... ma non dovrebbe accadere
     goto exit_func;
```

Creo la stringa (ossia i dati) dal valore di `.size` e calcolo qual è la lunghezza. Subito però controllo che offset non sia tale da puntare alla riga successiva. In questo caso offset va decrementato per tenere conto che ci si è spostati avanti per calcolare quanto rimane della seconda stringa. Chiaramente se sono sulla seconda riga non devo trasferire la prima, quindi si salta al trasferimento della seconda (indirizzo `next_value`).

```
   numlen=sprintf(num,"%u\n",device->size); // creazione stringa
   if(offset < numlen) // e verifica di non scrivere troppo
     i=offset;
   else {
     offset-=numlen; //decremento offset perchè
     numlen=sprintf(num,"%u\n",device->index);
     if(offset < numlen) {
        i=offset;
        goto next_value;
     } else
        goto exit_func;
   }
```

Se si è qui si procede con il trasferimento della prima stringa

```
    do { //trasferisco il primo numero
       if(len<count) // controllo massima lunghezza
          buf[len++]=num[i];
       else
         goto exit_func;
    } while(num[i++]!='\n'); // scrivere tutto fino all'ultimo carattere '\n'

    numlen=sprintf(num,"%u\n",device->index); //creo la seconda stringa
    i=0;
next_value:
```

Si procede alla trasmissione della seconda riga (ossia il secondo numero)

```
    do { //trasferisco il secondo numero
      if(len<count) // controllo anche la massima lunghezza
        buf[len++]=num[i];
      else
        goto exit_func;
    } while(num[i++]!='\n');
```

Quando si esce dalla funzione (`exit_func`) si pone subito `*eof=1` perchè questo indica alle funzioni in user-space che il trasferimento dati è concluso. Si noti che ciò è significativamente differente dall'implementazione delle normali funzioni POSIX di accesso al device driver.

```
exit_func:
    *eof=1;
    up(&device->sem);
    return len;
}
```

charbuf_write_procmem_dim

Questa prima write permette di effettuare la riconfigurazione delle dimensioni del buffer. Scrivendo 'x' seguito da un solo numero, moltiplico la dimensione del buffer (entro i limiti), mentre se al posto di 'x' vi è ':', allora si dividerà il valore. La funzione elabora solo i primi 2 caratteri (per esempio "x34\n"="x3") e il secondo deve essere un numero da 2 a 9.

```
int charbuf_write_procmem_dim(struct file *file, const char __user *buf,
                              unsigned long count, void *data) {
    int i, newbufdim=0;
    char b[2], *auxbuf=NULL;
    struct charbuf_dev *device=charbuf_device;

    if(down_interruptible(&device->sem))
      return -ERESTARTSYS;

    if(!buf) //probabilmente superfluo
      goto exit_func;

    if(count>=2)
      copy_from_user(b, buf, 2); //prendo solo i primi 2 byte.
    else
      goto exit_func;
```

La dimensione deve essere minimo di 2 caratteri perchè se potesse raggiungere il valore 0 o 1, a fronte di successive moltiplicazioni o divisioni, il buffer non si ridimensionerebbe più.

```
    if((newbufdim=b[1]-'0')>9) //massimo 9...
      goto exit_func;
    else if(newbufdim<2)
      goto exit_func;
```

Ridimensionando il buffer è necessario ricopiare i dati attualmente scritti tenendo conto del parametro `.index`. Chiaramente il buffer dovrà essere troncato se la nuova dimensione è più piccola di `.index`.

```
auxbuf=device->charbuffer; //"sposto" il buffer su un altro puntatore ausiliario
if(b[0]=='x') { //fase di zione
    device->size*=newbufdim;
     if(device->size>MAX_BUF_DIMENSION)
       device->size=MAX_BUF_DIMENSION;
     if(!(device->charbuffer=(char*)kmalloc(device->size,GFP_KERNEL))) {
          printk(KERN_ALERT "Not enought memory!\n");
          goto exit_func;
       }
     for(i=0; i<device->index; i++)
       device->charbuffer[i]=auxbuf[i];
  }
else if(b[0]==':') { //fase di divisione
     device->size/=newbufdim; //minimo devo mettere 1 byte
     if(device->size<=0)
       device->size=1; //minimo 1 byte
     if(!(device->charbuffer=(char*)kmalloc(device->size,GFP_KERNEL))) {
          printk(KERN_ALERT "Not enought memory!\n");
          goto exit_func;
       }
     for(i=0; i<device->size && i<device->index; i++)
       device->charbuffer[i]=auxbuf[i];
     device->index=i;
  }
printk(KERN_ALERT "Redimensioning buffer to %i byte through /proc\n", device->size);
```

In fase di chiusura libero il buffer ausiliare, faccio in modo che l'accesso al driver sia ripristinato anche per gli altri processi e infine restituisco `count` in modo tale che il programma in user-space sappia subito che i dati inviati sono stati tutti "recepiti" e quindi non proceda con altre chiamate della funzione

```
    exit_func:
    if(!auxbuf) kfree(auxbuf);
    up(&device->sem);
    return count; //restituisco count in modo da terminare sempre l'invio di dati
}
```

4.1.4 Funzioni per `buffer`

Le funsioni sono `charbuf_read_procmem_buf` e `charbuf_write_procmem_buf`.

charbuf_read_procmem_buf

Legge anche in questo caso la memoria del device ossia del buffer vero e proprio. Vale ovviamente tutto ciò che è stato detto per la funzione '`charbuf_read_procmem_dim`'.

La funzione deve prevedere l'accesso a programmi come *dd* i quali possono cominciare a leggere i file non dall'inizio, ma da un qualsiasi punto, ed inoltre può richiedere che il buffer in user-space sia caricato con blocchi di diversa dimensione (`count`). Per fare ciò occorre come noto indicare al programma quando è raggiunto l'eof, ma anche restituire il numero di byte trasferiti, ossia '`i`'(e non `len` come nella versione read descritta in 4.1.3). Come detto precedentemente, a differenza delle *read* normali, in questo caso è sufficiente `*eof=1` per indicare il terminamento del processo di riempimento del buffer `buf`, mentre nelle *read* normali (POSIX) il processo di lettura si arresta a fronte di un `return 0`. Ultima nota fondamentale è che per trasferire un buffer di dimensione X tramite n pacchetti più piccoli di dimensione Y, occorre aggiungere `*start=buf` che permette di usare una sorta di "paginazione" (vedere [1, cap4 pag84]).

```
int charbuf_read_procmem_buf(char *buf, char **start, off_t offset,
                             int count, int *eof, void *data) {
```

`len`, ossia `offset`, indica da che punto del buffer sorgente si comincia a leggere. `i` è un contatore, ma anche il ritorno della funzione che indica quanti byte sono stati copiati

```
  int len=offset, i;
  struct charbuf_dev *device = charbuf_device;
  if(down_interruptible(&device->sem))
     return -ERESTARTSYS;
  if(offset >= device->index) {
     *eof = 1; //segnalo che la parte utile del buffer è esaurita
     i = len;  //anche se non è vero, è bene passare questo valore di byte copiati
     goto exit_func;
  }
  if(count + offset >= device->index)
     count = device->index - offset; //qui offset è sicuramente < di .index

  *start=buf; //per lettura del buffer copiando piccole parti
  for(i=0; i<count; i++) //fase di copiatura dei singoli byte
     *(*start + i) = device->charbuffer[ len++ ];

  exit_func:
  up(&device->sem);    // nella versione 03 la riga sotto era 'return len;'
  return i; //restituisco quanti byte sono stati copiati
}
```

charbuf_write_procmem_buf

La funzione è molto semplice: se viene scritto 0 con

```
echo 0 > /proc/charbuf/buffer
```

il buffer si annulla (`memeset`). Si noti che alla fine viene restituito `count` per fare in modo che i programmi user-space vedano subito che tutti i dati sono stati "recepiti" ed elaborati. Ciò impedisce che il processo effettui nuovamente la chiamata di *write*.

```
int charbuf_write_procmem_buf(struct file *file, const char __user *buffer,
                              unsigned long count, void *data) {
   struct charbuf_dev *device = charbuf_device;
   char c;

   if(!count) //se è 0 non fare nulla
     return count;

   if(down_interruptible(&device->sem))
     return -ERESTARTSYS;

   copy_from_user(&c, buffer, 1);
   if(c=='0') { //annullo il buffer
        memset(device->charbuffer, 0, device->size);
        device->index=0;
        printk("Clean buffer (%i byte)\n", device->size);
     }

   up(&device->sem);
   return count; //tutto in una sessione
}
```

4.2 Test dell'interfacciamento di */proc*

Anche in questo caso ovviamente il modulo deve essere caricato. Il modulo avrà generato la directory virtuale */proc/charbuf*. Si verifica la presenza dei file con:

```
$ ls /proc/charbuf/
buffer  dimension  timerstr
```

Per procedere ai test si consideri di aver caricato il modulo definendo la dimensione del buffer a 64 byte e di riempire tale buffer con una ventina di caratteri:

```
# insmod charbuf.ko bufdimension=64
# echo 1234567890abcdefghi > /dev/cb0
```

dimension

Lettura di tutto il file:

```
$ cat /proc/charbuf/dimension
64
20
```

Lettura di una parte di file a partire dall'inizio, in particolare lettura di 2 byte letti uno alla volta:

```
dd if=/proc/charbuf/dimension bs=1 count=2
61+0 records in
1+0 records out
```

Il primo numero ('5') è il risultato effettivo a fronte di 51 come risultato atteso. Questo è corretto per i seguenti motivi: leggendo il codice, la funzione di lettura di dimension prevede `*eof=1` alla fine che, come detto, fa interrompere l'esecuzione del programma in user-space. A dire la verità però anche se questa istruzione non ci fosse, l'output non sarebbe diverso e non sarebbe corretto. Questo è dovuto al fatto che la trasmissione dei dati (in questo caso verso lo user-space) eseguita per piccoli passi va gestita con il puntatore `*start` come fatto nella funzione '`charbuf_read_procmem_buf`'. Questo vale anche per `offset` diverso da zero e ciò si verifica aggiungendo al comando *dd* il parametro *skip* per esempio `skip=1`; il risultato sarà di aver copiato 0 byte.

Scrivendo in *dimension* si ha la ridimensione. Per verificare basta leggere lo stesso file:

```
$ echo :8 > /proc/charbuf/dimension
$ cat /proc/charbuf/dimension
8
8
# cat /dev/cb0
12345678
$ echo x5 > /proc/charbuf/dimension
$ cat /proc/charbuf/dimension
40
8
```

Questo verifica che il ridimensionamento del file è corretto.

buffer

A questo punto il buffer è di 40 byte ed è stato scritto per 8 byte. Con un semplice *cat* si verifica visivamente l'uguaglianza dei dati, mentre i più paranoici potranno usare *md5sum*:

```
$ cat /proc/charbuf/buffer /dev/cb0
1234567812345678
$ md5sum /proc/charbuf/buffer /dev/cb0
25d55ad283aa400af464c76d713c07ad  /proc/charbuf/buffer
25d55ad283aa400af464c76d713c07ad  /dev/cb0
```

Si verifica ora la lettura di una sola porzione del buffer partendo da 0 e da un offset diverso da 0

```
$ dd if=/proc/charbuf/buffer bs=1 count=5 skip=0
123455+0 records in
5+0 records out
$ dd if=/proc/charbuf/buffer bs=2 count=2 skip=2
56782+0 records in
2+0 records out
```

Rimane quindi da testare la sola scrittura. Si ricorda che la funzione di scrittura azzara il file se si scrive il carattere '0' in */proc/charbuf/buffer*. Quindi si procede con

```
$ echo 0 > /proc/charbuf/buffer
$ cat dev
$ cat /proc/charbuf/dimension
40
0
```

unload

Scaricando il modulo deve scomparire la directory sottodirectory *charbuf* e i relativi file contenuti all'interno.

5 Timer & Scheduling

Il modulo fino ad ora scritto ha permesso di capire quali sono i meccanismi di base che reggono la gestione di un device file di tipo char. Ora sfruttando questo modulo è possibile studiare alcuni metodi di temporizzazione offerti dal kernel. Ciò è in linea con quanto ci si è preposti all'inizio, ossia realizzare un driver indipendente dall'architettura, in quanto ora verranno testate solo funzioni messe a disposizione del kernel.

La base della temporizzazione del kernel è la variabile `jiffies`. Questa variabile esiste sia nella forma a 32 che 64 bit ed è definita in *linux/jiffies.h* come

```
extern u64 __jiffy_data jiffies_64;
extern unsigned long volatile __jiffy_data jiffies
```

Questa variabile non è altro che un contatore che viene incrementato ad una frequenza definita dalla macro `HZ`. Questa macro viene definita in fase di configurazione del kernel e tipicamente può essere settata a 100/250/1000 Hz. Per settarla è sufficiente lanciare la configurazione del kernel (`# make menuconfig`) e spostarsi nella sezione "Processor type and features —>" alla voce "Timer frequency" come riportato in figura 1.

Senza entrare nel dettaglio di cosa comporta il valore questa macro, basti pensare che in linea di massima ad essa è imputata parte della schedulazione dei processi, ma la sua funzione più importante è appunto quella di "comunicare" al processo quanto tempo è trascorso. In altre parole `jiffies` è un vero e proprio timer a livello kernel. Per maggiori informazioni si rimanda a [2, Cap 6 e 7]. L'introduzione di questa variabile è necessaria, come si vedrà, perchè è normalmente necessario salvare il valore istantaneo del timer per procedere con le temporizzazioni.

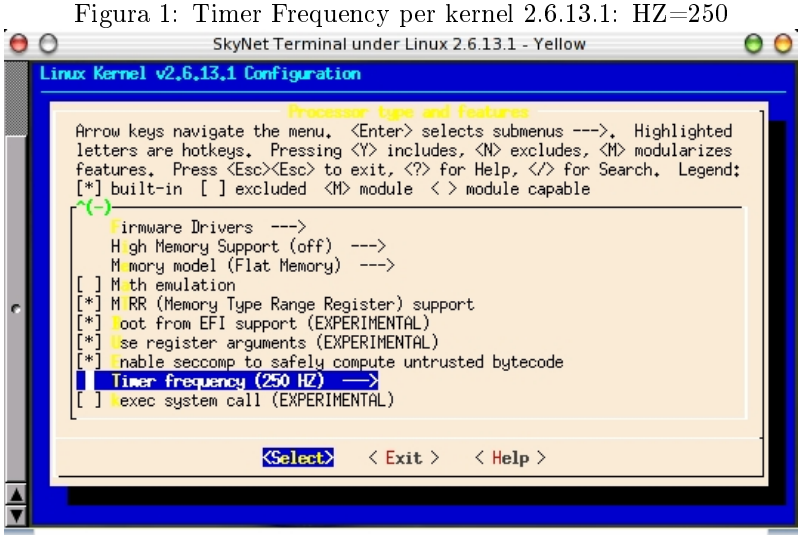
Rimane ora da definire quali funzioni usare per temporizzare/schedulare il processo. Sono previste 5 modalità di funzionamento definite dalle macro `TOD_`x dove x è un numero da 0 a 4; TOD è un acronimo che intica *Type of Delay*. Ma come fare a testarle? L'idea proposta è quella di creare un file virtuale in */proc*, più precisamente */proc/charbuf/timerstr*, che passa i vari caratteri di una stringa a intervalli regolari. Per settare la modalità di ritardo (TOD appunto) bisogna scrivere nel file *timerstr* un valore da 0 a 4. Le funzioni richiamate dal modulo in funzione del valore di TOD sono:

`TOD_0` Non fa nulla; trasferisce la stringa direttamente come se fosse un normalissimo buffer.

`TOD_1` `cpu_relax()`

`TOD_2` `schedule()`

Figura 1: Timer Frequency per kernel 2.6.13.1: HZ=250



`TOD_3` `init_waitqueue_head()` e `wait_event_interruptible_timeout()`

`TOD_4` `set_current_state()` e `schedule_timeout()`

Il modulo imposterà la funzione 4 di default. Segue ora l'implementazione delle funzioni.

5.1 Funzioni per timerstr

Le funzioni sono `charbuf_write_procmem_dim` e `charbuf_read_procmem_tmrstr`.

charbuf_write_procmem_dim

Questa funzione elabora il primo carattere de buffer inviato da user-space e se è un carattere tra 0 e 4, verrà impostata la nuova modalità TOD.

```
int charbuf_write_procmem_tmrstr(struct file *file, const char __user *buffer,
                                 unsigned long count, void *data) {
   struct charbuf_dev *device = charbuf_device;
   char c;

   if(!count) return count;

   if(down_interruptible(&device->sem))
     return -ERESTARTSYS;

   copy_from_user(&c, buffer, 1);
   if(( c >= '0' ) && ( c <= '4' )){
      type_of_delay = c - '0';
      printk(KERN_ALERT "Set type of delay: %d\n", type_of_delay);
   }

   up(&device->sem);
   return count; //return count per fare tutto in una sessione
}
```

charbuf_read_procmem_tmrstr

Questa funzione deve permettere la lettura del file da programmi come *cat* e *dd*. Agli effetti esterni la funzione deve trasferire un carattere ogni 0.1 secondi (`JIFFIES_DEC`).

```
int charbuf_read_procmem_tmrstr(char __user *buf, char **start, off_t offset,
                                int count, int *eof, void *data) {
   int len=offset, i=0; //len=0 chiude la procedura di lettura
   const char bufstr[] = CHARBUF_PROC_STR;
   int j1;
   wait_queue_head_t wait;

   if (offset >= (sizeof(CHARBUF_PROC_STR))-1) { //if precauzionale
      *eof = 1; //Nei file virtuali occorre sempre segnalare l'eof
      i = len;  //FONDAMENTALE per dd. NOTA: len=ofset
      goto exit_func;
   }
   if ( count+offset >= (sizeof(CHARBUF_PROC_STR))-1 )
     count = sizeof(CHARBUF_PROC_STR)-offset-1;
```

si ricorda che `*start` deve essere usato quando il programma in user-space rchiede più volte consecutivamente l'accesso alla funzione di lettura

```
   *start=buf; //start serve percè se scrivo "+ pagine" non posso usare buf,
   switch ( type_of_delay ) {
```

TOD_0 non realizza nessuna temporizzazione. Il riempimento del buffer user-space avviene con `*start`

```
case TOD_0: // nothing
  for (i=0; i < count; i++)
    *(*start+i) = *(bufstr+len+i);
  break;
```

TOD_1 temporizza il processo tramite `cpu_relax()`. Questa funzione è strettamente dipendente dalla architettura. Normalmente comunque cerca di portare la CPU in idle.

```
case TOD_1: // delay with cpu_relax()
  for (i=0; i < count; i++) {
       j1=jiffies + JIFFIES_DEC; //leggo e salvo i jiffies "futuri"
        *(*start+i) = *(bufstr+len+i);
       while (time_before(jiffies, j1))
         cpu_relax(); // funzione strettamente legata all'architettura
    }
  break;
```

TOD_2 salva i `jiffies` attuali sommando una quantità di jiffies pari a `JIFFIES_DEC` per attendere 0.1s tra la trasmissione di un carattere e l'altro. Successivamente tramite la funzione `time_before` un ciclo while richiede la schedulazione continua del processo. Ciò implica che se ci fosse un processo (nella coda dei processi) di priorità maggiore o uguale del processo corrente che ha richiesto la rischedulazione, il processo corrente (ossia quello che sta leggendo *timerstr*) viene interrotto per eseguirne un altro.

```
case TOD_2:
  for (i=0; i < count; i++) {
       j1=jiffies + JIFFIES_DEC; //leggo e salvo i jiffies "futuri"
       *(*start+i) = *(bufstr+len+i);
       while (time_before(jiffies, j1))
         schedule();
    }
  break;
```

TOD_3 è un modo più elegante di gestire il processo. In questo caso non vengono salvati i jiffies, ma viene inizializzata una variabile (tramite `init_waitqueue_head`) che permetterà di gestire l'evento di attesa. Scaduta l'attesa il processo interrotto viene riattivato dal kernel. Il processo durante la fase di attesa viene configurato come "interruttibile" il che implica che in caso di "morte" del processo, tale processo può essere eliminato dalla coda dei processi.

```
case TOD_3:
  for (i=0; i < count; i++) {
       init_waitqueue_head (&wait); // inizializzo la "coda" degli eventi di attesa
       *(*start+i) = *(bufstr+len+i);
       wait_event_interruptible_timeout(wait, 0, JIFFIES_DEC); // aspetto
    }
  break;
```

TOD_4 è ancora più efficiente di TOD_3. In questo caso si usa `sched_timeout` andando a settate il task corrente in modalità interruttibile, per gli stessi motivi descritti a TOD_3. Questa volta però il task viene rischedulato.

```
case TOD_4:
  for (i=0; i < count; i++)
    { //alcune opzioni (de sched.h): TASK_UNINTERRUPTIBLE, TASK_RUNING
       set_current_state(TASK_INTERRUPTIBLE);
       *(*start+i) = *(bufstr+len+i);
       schedule_timeout (JIFFIES_DEC);
       set_current_state(TASK_RUNNING); // questo probabilamente non serve
    }
  break;

default:
  printk(KERN_ALERT "Warning: invalid delay option!\n");
```

```
            break;
        }

exit_func:
   return i; //devo restituire count, ossia il numero di byte copiati
}
```

5.2 Test del timer

Il test verrà fatto su kernel 2.6.13.1 con kernel preemptive e timer a 250Hz. La stringa che timerstr deve trasferire è definita dalla macro `CHARBUF_PROC_STR` pari a `"CharBuf: modulo indipendente dall'architettura\n\t--\033[31mby Calzo\033[0m --\n"`. I caratteri di escape creeranno una parte della stringa scritta in rosso. Il risultato è il seguente:

```
CharBuf: modulo indipendente dall'architettura
        -- by Calzo --
```

TOD_0 I comandi eseguiti settano TOD=0 e leggono *timerstr* in due modalità misurandone il tempo di esecuzione

```
# echo 0 > /proc/charbuf/timerstr
# time cat /proc/charbuf/timerstr
# time dd if=/proc/charbuf/timerstr bs=1
```

Il risultato è sempre di 0.002s.

TOD_1

```
#echo 1 > /proc/charbuf/timerstr
#time cat /proc/charbuf/timerstr
real    0m7.200s
user    0m0.000s
sys     0m7.184s
```

Tramite *cat* occorre aspettare 7.2s prima di vedere visualizzata la stringa; ciò è corretto perchè i caratteri da trasferire sono 72 che a 0.1s ciascuno sono 7.2s.

```
#time dd if=/proc/charbuf/timerstr bs=1
real    0m7.198s
user    0m0.000s
sys     0m7.156s
```

Similmente *dd* risponde nello stesso modo. Si noti che *bs=1* non fa raggiungere esattamente i 7.2s; se fosse *bs=72* o maggiore il comportamento sarebbe uguale a *cat*.

La cosa particolare è che di questo 7.2s, quasi tutto il tempo rimane in spazio kernel! Si noti poi che durante l'esecuzione non sono stati lanciati altri programmi, quindi tutta la gestione è in mano allo scheduler che comunque non esegue nulla di significativo, tranne se stesso; ciò dovrebbe spiegare come mai il tempo di sistema sia sempre inferiore a quello reale. Per verificare l'affermazione appena fatta è sufficiente lanciare contemporaneamente due *cat*:

```
#time cat /proc/charbuf/timerstr & time cat /proc/charbuf/timerstr
real    0m13.684s
user    0m0.000s
sys     0m6.828s
real    0m13.757s
user    0m0.000s
sys     0m13.753s
```

Come si vede i tempi di sistema e quelli reali cambiano nettamente, come era prevedibile in quanto, se si esclude la rischedulazione (che dovrebbe essere richiesta da `cpu_relax()`), i processi entrati in kernel è come se eseguissero un loop in attesa dello scadere del tempo. Sembrerebbe quindi che il sistema non esegua ciò che si desidera, ma in realtà non è così; il fatto che i tempi si allunghino dovrebbe essere dovuto al fatto che i processi hanno pari priorità e, mentre "aspettano", sono processi in spazio kernel. Si noti infatti che portando il carico del processore vicino al 100%, il tempo di kernel viene nettamente ridotto a vantaggio del processo user-space che praticamente non risulta essere penalizzato. Per testarlo si lancino in due differenti shell i comandi seguenti:

```
$ while true; do echo -n X; done   # prima shell
# time cat /proc/charbuf/timerstr  # seconda shell
real    0m7.205s
user    0m0.000s
sys     0m0.004s
```

TOD_2 Il secondo metodo implementato è praticamente identico al precedente, cosa intuibile visto che anche in questo caso la struttura ti temporizzazione si appoggia ad un ciclo *while* dove viene subito chiamato la funzione `schedule()`; questa funzione è lo scheduler vero e proprio ed è abbastanza complicata. Per chi volesse approfondire si rimanda a [2, cap7 - Process Scheduling], ma per quanto concerne il modulo, basti sapere che la funzione controlla il processo corrente e, se necessario (ossia nella maggior parte dei casi) lo interrompe in favore di un processo con priorità maggiore o uguale.

Sia *cat* che *dd* restituiscono gli stessi valori di TOD_1. Le cose cambiano se vengono lanciati due processi contemporaneamente:

```
# echo 2 > /proc/charbuf/timerstr
# time cat /proc/charbuf/timerstr & time cat /proc/charbuf/timerstr
real    0m12.953s
user    0m0.000s
sys     0m6.408s
real    0m13.558s
user    0m0.000s
sys     0m7.096s
```

in questo caso il tempo di kernel viene suddiviso più equamente rispetto a prima.

TOD_3

```
# echo 3 > /proc/charbuf/timerstr
# time cat /proc/charbuf/timerstr
real    0m7.200s
user    0m0.000s
sys     0m0.000s
```

I dati riportati sono i migliori risultati ottenuti; si noti infatti che se ci fossero più processi in esecuzione il tempo reale potrebbe variare di qualche ms. Ad ogni modo il dato importante è che il tempo di sistema è pressochè nullo. Analogamente anche per *dd*:

```
# time dd if=/proc/charbuf/timerstr bs=1
real    0m7.199s
user    0m0.000s
sys     0m0.000s
```

Le prestazioni massime si osservano in multi-process. Lanciando due processi si ha:

```
# time cat /proc/charbuf/timerstr & time cat /proc/charbuf/timerstr
real    0m7.199s
user    0m0.000s
sys     0m0.000s
real    0m7.202s
user    0m0.000s
sys     0m0.004s
```

Il processore resta scarico e i tempi sono perfettamente rispettati.

TOD_4 Dal punto di vista prestazionale la lettura di */proc/charbuf/timerstr* è identico.

A CRIS

Per testare il modulo sotto architettura CRIS viene utilizzata la scheda FOX Board. Per informazioni più dettagliate sullo sviluppo di un modulo per questa scheda si rimanda a [3]. Ad ogni modo basti sapere che per la compilazione del modulo trattato in questo articolo, i sorgenti sono stati copiati in una directory apposita. La directory in questione è *drivers/calzo/* nella root del kernel CRIS. Questa directory va aggiunta a `$AXIS_KERNEL_DIR`*/arch/cris/Kconfig*. A questo punto è successa una cosa molto strana: il nome del file *charbuf.c* da dei problemi alla compilazione la quale crea solo il file oggetto, ma non il morulo. Quindi il file verrà rinominato in *cris_charbuf.c*. Il Makefile in *calzo/* semplicemente includerà come modulo *cris_charbuf* con la riga `obj-m += cris_charbuf.o`.

A questo punto per compilare il modulo basta lanciare *make* (*make modules*) nella root del kernel dopo aver configurato l'ambiente di sviluppo lanciando in `$AXIS_KERNEL_DIR/` il comando "`. init`". La compilazione va subito a buon fine, segno che il modulo è portabile senza modifiche. Ciò era prevedibile in quanto non è stata utilizzata nessuna istruzione dipendente dall'architettura direttamente nel modulo. Funzioni come `schedule()` o `cpu_relx()` sono sì dipendenti dall'architettura, ma fornite con il kenrnel e quindi programmando a questo livello non si hanno problemi.

Una volta compilato il modulo, lo si copia nella FOX, ci si collega e lo si carica:

```
scp cris_charbuf.ko root@192.168.0.90:/mnt/flash
telnet 192.168.0.90
insmod cris_charbuf.ko
```

A questo punt il modulo è caricato e si possono rieseguire gli stessi test fatti in precedenza. Si noti che il kernel 2.6.x della FOX Board può essere configurato per essere un kernel preemptive (e normalmente lo è), può essere configurato il big-kenrel-look, ma la frequenza del timer è fissa a 250Hz[2].

Si noti che la configurazione di default della FOX non include il comando time, quindi non sarà possibile testare il modulo senza modificare il sistema e/o il modulo. Si noti inoltre che il sistema in questo caso porta tutti i messaggi del kernel (`KERN_INFO`, `KERN_ALERT`, ecc) vengono inoltrati a */var/log/messages*.

B How to compile a module

Per un kernel x86 è possibile ompilare un modulo da qualsiasi directory. É sufficiente quindi scrivere un *Makefile* con le regole necessarie. Il kenrel in questione è un kenrel 2.6, quindi la struttura del file può essere la seguente:

```
ifneq ($(KERNELRELEASE),)
        obj-m := charbuf.o
else
    KERNELDIR ?= /lib/modules/$(shell uname -r)/build
    PWD := $(shell pwd)

default:
        $(MAKE) -C $(KERNELDIR) -I $(KERNELDIR)/include M=$(PWD) modules
clean:
        rm *.ko
        rm *.o
        rm .charbuf*
        rm -r .tmp_versions
endif
```

Così facendo verrà creato un modulo chiamato *charbuf.ko*. Per includere il modulo nel kernel occorre porre il Makefile nel kernel tree e configurare i file *Kconfig* opportunamente. Per vedere come si rimanda a [3, Appendice A].

[2]Sarebbe possibile modificarla, ma solo manualmente nei file di configurazione; si consiglia comunque di non aumentarla.

Riferimenti bibliografici

[1] *Linux Device Driver 3th Edition* - Jonathan Corbet, Alessandro Rubini and Greg Kroah-Hartman - O'REILLY

[2] *Understanding The Kernel Linux 3th Edition* - Daniel P. Bovet and Marco Cesati - O'REILLY

[3] *FoxDoc.pdf* - Documento relativo alla programmazione user/kernel-space della FOX Board per il controllo delle GPIO - scaricabile dal sito www.lugman.org

[4] http://www.faqs.org/docs/kernel/index.html - Esempio di scrittura di un modulo per il kernel

Info & Credits

Articolo scritto e presentato al LINUX DAY 2008 - ottava giornata di Linux e del Software Libero - dall'associazione LUGMan (Linux Users Group Mantova). L'articolo è scaricabile in formato PDF ed è distribuito insieme ai sorgenti del modulo escritto. Il tutto è scaricabile liberamente dal sito dell'associazione (www.lugman.org nella sezione "Documentazione"). L'articolo è stato scritto con LyX 1.4.3 sotto Slackware 10.2 da Calzoni Pietro aka *Calzo*.

Chiunque volesse altri formati del documento o i sorgenti e non riuscisse a trovarli in altro modo, può richiederli contattando l'associazione a info@lugman.org.

Chiunque volesse contattare l'associazione o l'autore può consultare il sito www.lugman.org nella sezione "Contatti" o scrivere a info@lugman.org.